

Programs as Proofs

Jørgen Steensgaard-Madsen
(Retired)

Abstract

The Curry-Howard correspondence is about a relationship between types and programs on the one hand and propositions and proofs on the other. The implications for programming language design and program verification is an active field of research.

Transformer-like semantics of internal definitions that combine a defining computation and an application will be presented. By specialisation for a given defining computation one can derive inference rules for applications of defined operations.

With semantics of that kind for every operation, each application identifies an axiom in a logic defined by the programming language, so a language can be considered a theory.

1 Introduction

The Curry-Howard correspondence relates programming languages and logic. Essentially it reflects similarity between propositions and proofs of a formal logic and types and programs of a programming language. W.A.Howard is the author of the seminal paper on the correspondence. Various researchers have investigated its implications for programming. Transformer semantics is just for a type of language design based on notions that relates to *weak existence* described by Howard separately in the same paper.

According to the Curry-Howard correspondence, the notions of programs correspond to proofs, and a program proves its type according to the typing rules of the programming language. So a powerful notion of types is needed for this to be of interest. Per Martin Löf [4] has advocated a system with such a type system.

Dijkstra's predicate transformers can be molded as a logic, but the predicates in such a system do not correspond to types, but programs correspond to proofs, if their applications are seen as recordings of proof steps. Dijkstra's language statement constructs have the trivial type *void* but in a language with a richer type system, typing rules would imply that programs seen as proofs would also prove the type of its result.

A formal system which includes expressions of the form $\exists_P \theta. R_\theta$ will be presented. These are mathematical expressions that combine the following parts: a program text P , a mathematical expression R_θ , and an operator $\exists_P \theta.$ which might be compared to the integration operator $\int _ d\theta$. The operator $\exists_P \theta.$ binds free occurrences of θ in the expression R_θ (as in lambda-calculus, say). The intuitive meaning of the abstraction variable is *an expression for the value obtained by interpretation of P* . $\exists_P \theta.$ is an automorphism on mathematical expressions, partly related to Dijkstra's predicate transformers.

The intuitive reading of $\exists_P \theta. R_\theta$ is that *P proves that an expression θ exists such that R_θ 'is of interest'*. The objective with $\exists_P \theta. R_\theta$ is similar to Dijkstra's, i.e. to obtain a

mathematical expressions that, in the state from which P should be interpreted, has the value of R_θ in the in the result state.

The verb ‘to prove’ is transitive, but it is possible to conceive the noun ‘proof’ independently of an object as required by the verb. A proof can be understood as a pattern of references to individual rules of a formal logic. A programmer writes programs that can be seen as proofs, when an adequate language is used. Programmers do not need to master the logic aspects, but provide proofs much like people may do in every-day arguments. Such proofs may need be checked but, hopefully, may be automated some day.

A benefit of programs as proofs is that not all programmers need to master the logic aspects of programs completely. One may still acknowledge programmers work as proofs in a substantial sense. Language designers should, of course, be more concerned with the rules for application of their operations, i.e. their adequateness as axioms of a logic.

Another benefit might be the implicit enforcement of a programming discipline that is indirectly influenced by formal logic, much the same as discussions in daily life may benefit from rules justified by formal logic.

2 Transformation semantics

Dijkstra has presented a notion of *predicate transformers* along with a simplified programming language, and thus illustrated an important relationship between language design and a logic for program proofs: [2]. However, transformation rules appear to arise as axioms from the intuition of the language designer. Dijkstra’s language does not support definitions in programs. But they are needed in practice and corresponding transformation rules likewise.

Rules for definitions within programs, and even inside definitions will be essential also to find rules for application of defined entities. Such definitions will be called *internal definitions*. It will be possible to write them in familiar style, but conceptually they differ a bit: a definition is more like familiar (letrec- or) let-constructs, i.e. it combines one part that constitutes a conventional definition with an application part where the defined entity can be applied.

The notion of internal definitions in an adequate programming language that admits P in $\exists_P \theta$. should allow definition of object-like entities, i.e. means to introduce ‘members’ associated with an ‘object’ of some ‘class’. An essential aspect is that programmers do not chose names of members, but accept names determined by the ‘class’.

Such introduction of names can be generalised, and the term *implicit name binding* will be used for it as a design principle. Furthermore, ‘in-line definitions’ should be mandatory, which can be done by admitting first-order types only. Higher-order entities will be admitted in another way. In other words: operation names may not be used as arguments and implicit name binding thus required for every ‘in-line definition’.

An internal definition can be compared to a class with the defined entity as its only member in the application part. Both notions combine details of semantics with a notion of abstraction over its application. Simula 67, [1], provides the *inner* notion for the ‘remaining

part of the current block’, like the scope of definitions in several other languages, e.g. Pascal and C. Simula’s *inner* notion will be replaced by a (named) parameter, and the idea is generalised so that implicit name binding may occur in several arguments.

An essential difference for programmers, between classes and internal definitions, is the required description of ‘parameters’ in the latter. Identification of an internal definition includes the parameter description, over which one cannot abstract. Parameter descriptions will be structured like those of Pascal [7], but extended with type parameters. Higher-order parameters are then described by the same syntax as used for internal definitions. The term *signature* will be used for parameter descriptions.

3 A combinator for internal definitions

Several languages support a simple construct

$$\text{let } f(x) = Expr_1 \text{ in } Expr_2$$

where the name f can be used in $Expr_2$ for a function that maps x to $Expr_1$. It can be explained reasonably well with lambda-notation by $(\lambda f. Expr_2)(\lambda x. Expr_1)$, i.e. an expression that can be rephrased as follows:

$$(\lambda B. \lambda A. (A(B))) (\lambda x. Expr_1) (\lambda f. Expr_2)$$

With no name occurring free, the expression $\lambda B. \lambda A. (A(B))$ is a *combinator* and can be given a global name, e.g. DEF:

$$\text{DEF } (\lambda x. Expr_1) (\lambda f. Expr_2)$$

which, except for a missing signature and the explicit introduction of names, forms an internal definition. A single argument combinator is appropriate for a *global definition* in agreement with mathematics, but not for internal definitions. This and the need to structure programs for human readers justifies internal definitions as a separate notion.

Lambda-calculus prescribe rewriting rules for expressions in lambda-notation. Here we just depend on the β -rule:

$$(\lambda x. Expr_1)(Expr_2) = [x \triangleleft Expr_2] Expr_1$$

where the right-hand side prescribes substitution of $Expr_2$ for x in $Expr_1$. However, a reservation is necessary to avoid unintended name binding, i.e. a free occurrence of a name in $Expr_2$ unintentionally getting bound in the process. A request for a name change in such situations is a typical way to state the reservation. Our use of the substitution symbol, \triangleleft , shall include this reservation.

The following simplified example illustrates DEF and use of the β -rule. However, it also illustrates that lambda-calculus is unfit to render the proceedings in a way that is easy for humans to follow. The many lambda-abstraction is one essential cause. Another is that equational reasoning here blends the contexts of the two arguments where they should be kept apart to help readers,

```

DEF
  (λ D λ A . A (λ X . D (X*2)))
  (λ r . r (λ x . (x*x)) (λ f . (f(3))))
=   (λ r . r (λ x . (x*x)) (λ f . (f(3))))
      (λ D . λ A . A (λ X . D (X*2)))
=   (λ D . λ A . A (λ X . D (X*2)))
      (λ x . (x*x)) (λ f . (f(3)))
=   (λ f . (f(3))) (λ X . (λ x . (x*x)) (X*2))
=   (λ X . (λ x . (x*x)) (X*2)) (3)
=   (λ x . (x*x)) (3*2)
=   (3*2)*(3*2)
=   36

```

Notice that the character case of names in the two arguments of DEF differ: all capitalised in the first, none in the second. It reflects a simple ping-pong-like game of control.

A programming language will be introduced in which an internal definition for the above can be expressed as:

```

DEF r OF T [D[x:int]:int1] [A[f[X:int1]:int]:T]: T
  { A{D{X*2}} }
  { r{x*x}{f{3}} };

```

The text after DEF on the first line is a *signature* and contains enough information to avoid explicit parametrisation (i.e. the λ-s). The third line illustrates use of the defined operation **r** which itself has a structure similar to an internal definition: its first argument defines a function that is required to derive the function **f** used in the second argument.

From a signature one can automatically derive a rule for application of the derived operation, i.e. **r** above:

$$\begin{aligned}
& \exists_{r\{E_D\}\{E_A\}} \theta. = \exists_{D_r} \theta. \\
& \neg \left[\begin{array}{l} \exists_{D\{D_x\}} \theta. = \exists_{E_D} \theta. \neg \left[\exists_x \theta. = \exists_{D_x} \theta. \right] \\ \exists_{A\{D_f\}} \theta. = \exists_{E_A} \theta. \neg \left[\exists_{f\{E_X\}} \theta. = \exists_{D_f} \theta. \neg \left[\exists_x \theta. = \exists_{E_X} \theta. \right] \right] \end{array} \right]
\end{aligned} \tag{1}$$

where D_r is the first argument of the internal definition. Before we see how such a rule helps to derive the result value we need to explain the structure.

Rules have the form of equations or *conclusion* \neg *presumptions* and are displayed as:

$$\exists_{\text{Left}} \theta. = \exists_{\text{Right}} \theta. \neg \left[\begin{array}{l} \text{presumption}_1 \\ \text{presumption}_2 \\ \dots \end{array} \right]$$

Each presumption can be a rule. Presumptions do hold by assumption.

Box 1 Axiom schemes

1. DEF-scheme

$$\exists_{\text{DEF } \mathbf{f} \dots \{D_{\mathbf{f}}\}\{E_{\mathbf{f}}\}} \theta. = \exists_{E_{\mathbf{f}}} \theta. \dashv \left[\mathbf{f} [P_i \dots]_{i=1..N_f} \stackrel{\text{def}}{=} \{D, E\} \right]$$

2. Annihilate

$$\exists_{\{\}} \theta. = [\theta \triangleleft \bullet] \quad \text{i.e. error if } \theta \text{ occurs free in the (omitted) operand}$$

3. Constant

$$\exists_{\mathbf{C}} \theta. = [\theta \triangleleft C] \quad \text{with } \mathbf{C} \text{ a constant}$$

4. Subsumed name translation (i.e. used only without an explicit alternative)

$$\exists_{\mathbf{X}} \theta. = [\theta \triangleleft X] \quad \text{NB: different fonts for } X$$

5. Sequential computation

$$\exists_{E_1; E_2} \theta. = \exists_{E_1} \theta. \exists_{E_2} \theta.$$

6. Infix operator symbols (with # varying over operator symbols)

$$\exists_{E_1 \# E_2} \theta. = \exists_{E_1} \tau_1. \exists_{E_2} \tau_2. [\theta \triangleleft (\tau_1 \otimes \tau_2)] \quad \text{where } \otimes \text{ identify the meaning of } \#$$

4 Formalisation of copy-rule semantics

Semantics of internal definitions is formalised by a scheme of transformation rules that essentially formalises the copy-rule semantics of Algol 60 [5]. The formalisation will be called the *DEF-scheme* for brevity and it constitutes the only complex transformation rule that serves as an axiom. Copy-rule semantics for Algol 60 has been described intuitively in a few words, but its formalisation is complex.

A version that covers only call-by-name parameters and with optional result types disregarded, forms an introduction to the general scheme.

$$\begin{aligned} \exists_{\text{DEF } \mathbf{f} \dots \{D_{\mathbf{f}}\}\{E_{\mathbf{f}}\}} \theta. &= \exists_{E_{\mathbf{f}}} \theta. \\ \dashv \left[\mathbf{f} [P_i \dots]_{i=1..N_f} \stackrel{\text{def}}{=} \{D, E\} \right] \end{aligned} \quad (2)$$

Symbol $\stackrel{\text{def}}{=}$ identifies a macro that combines a signature and a switch for control status, $\{D, E\}$, which indicates alternation between definition- and application-contexts. This scheme is the interesting axiom of a logic of transformations. Box 1 presents all of them.

The rules allow side-effects and imply call-by-name. Call-by-value can be covered as expressed for application of an operation with signature

with OF T, W (X:T) [Body(__:T):W] : W

and semantics

$$\begin{aligned} \exists_{\text{with}\{E_X\}\{E_{\text{Body}}\}} \theta. &= \exists_{E_X} \eta_1. [X \triangleleft \eta_1] \exists_{D_{\text{with}}} \theta. \\ \dashv \left[\exists_{\text{Body}\{D_{-}\}} \theta. &= \exists_{D_{-}} \eta_1. [_{-} \triangleleft \eta_1] \exists_{E_{\text{Body}}} \theta. \right] \end{aligned}$$

Parentheses can be used in signatures to request call-by-value. The example illustrates the obvious: that the axiom schemes above do not suffice, and that substitution is not not a trivial operation.

The form of internal definitions can be described with EBNF as

“DEF” *signature* “{” D_- “}” “{” E_- “}
 where *signature* = *name* { “[” *signature* “]” }_{1..N} [: *type*] and $N \geq 0$

Signatures will be described in greater details later.

The DEF-scheme in Equation 2 tells that a definition of f combines two arguments D_f and E_f , and that the entire construct as one expression is given by $\exists_{E_f} \theta.$ with a presumption about applications of f .

Presumptions in the DEF-scheme tell that application of a name in one context is combined with a defining argument in the other. This combination represents the formalisation of the copy-rule, with their environment represented by presumptions.

The complication of the DEF-scheme is expressed by the macro $- \stackrel{\text{def}}{=} \{-, -\} :$

$$\begin{aligned} f [P_i [g_{i,j} \dots]_{j=1..n_i}]_{i=1..N_f} &\stackrel{\text{def}}{=} \{D, E\} \\ &\equiv \exists_{f\{E_{P_i}\}_{i=1..N_f}} \theta. = \exists_{D_f} \theta. & N_f \geq 0 \\ &\quad \neg \left[P_i [g_{i,j} \dots]_{j=1..n_i} \stackrel{\text{def}}{=} \{E, D\} \right]_{i=1..N_f} \end{aligned} \quad (3)$$

A tool can generate DEF-schemes from signatures. Even from more complex signatures that may contain call-by-value parameters, type expressions, and (possibly overloaded) signatures of operator symbols.

A transformation rule for internal definitions is fairly complex. A one-step expansion of the def-macro in the presumption of Equation 3 may bring some relief:

$$\begin{aligned} f [P_i [g_{i,j} \dots]_{j=1..n_i}]_{i=1..N_f} &\stackrel{\text{def}}{=} \{D, E\} \\ &\equiv \exists_{f\{E_{P_i}\}_{i=1..N_f}} \theta. = \exists_{D_f} \theta. & N_f \geq 0 \\ &\quad \neg \left[\begin{aligned} &\exists_{P_i} \{D_{g_{i,j}}\}_{j=1..n_i} \theta. = \exists_{E_{P_i}} \theta. \\ &\neg \left[g_{i,j} \dots \stackrel{\text{def}}{=} \{D, E\} \right]_{j=1..n_i} \end{aligned} \right]_{i=1..N_f} \end{aligned} \quad (4)$$

Equation 4 expresses a presumption for application of DEF, which is a rule for applications of f . That rule is stated in terms of the unknown D_f , the first argument of DEF, the presumptions of which are similar rules for application of operations P_i . Applications of P_i is stated in terms of unknowns E_{P_i} (i.e. arguments of f). The presumption of E_{P_i} are rules for applications of operations g_j (i.e. implicitly introduced names) in terms of unknowns D_{g_j} .

Eventually a rule reduces to a pattern of substitutions corresponding to:

$$\left[f \triangleleft \lambda P_1. \lambda P_2. \dots \lambda P_{N_f}. D_f \right] E_f \quad \left[P_i \triangleleft \lambda g_1. \lambda g_2. \dots \lambda g_{n_i}. E_{P_i} \right] D_f \quad (5)$$

The objective has been to obtain rules that do not use lambda-abstractions, but rather point-wise use of functions and implicitly introduced names. Reservations about unintended name bindings associated with \triangleleft apply for \neg also.

An important issue related to Equation 3 is whether some language may satisfy the property. To see that, we characterise a language as follows

- Every operation must have a signature
- Application expression of the language has the form $f\{E_1\}\{E_2\}...\{E_{N_f}\}$ when the signature of f is

$$\begin{array}{ccccccc} f & [P_1 & [g_{1,1} & \dots] & [g_{1,2} & \dots] & \dots & [g_{1,n_1} & \dots]] \\ & [P_2 & [g_{2,1} & \dots] & [g_{2,2} & \dots] & \dots & [g_{2,n_2} & \dots]] \\ & \dots & & & & & & & \\ & [P_{N_f} & [g_{N_f,1} & \dots] & [g_{N_f,2} & \dots] & \dots & [g_{N_f,n_{N_f}} & \dots] &] \end{array}$$

where $N_f \geq 0$ and $n_k \geq 0$

- Each application expression is interpreted as a lambda-expression

$$\mathcal{T}(f\{E_1\}\{E_2\}...\{E_{N_f}\}) = f(\underline{\lambda g_1}.\mathcal{T}(E_1))(\underline{\lambda g_2}.\mathcal{T}(E_2))\dots(\underline{\lambda g_{N_f}}.\mathcal{T}(E_{N_f}))$$

where $\underline{\lambda g_k}$ stands for $\lambda g_{k,1}.\lambda g_{k,2}.\dots.\lambda g_{k,n_k}$

\mathcal{T} introduces lambda-abstractions for implicitly introduced names and in agreement with the bindings expressed in Equation 5

- Each internal definition **DEF** *signature* $\{D\}\{E\}$ is an application of combinator **DEF** equipped with signature

$$\mathbf{DEF}_{signature} \ [signature] \ [App[signature] \]$$

Let this language be called \mathcal{L} . The terms *Howard languages* and *Howard programs* will be used for languages characterised like \mathcal{L} and their programs, respectively.

PROPOSITION 1 *Language \mathcal{L} satisfies Equation 2-4, if adequate means ensure that names are distinct.*

PROOF. The interpretation \mathcal{T} implies that \mathcal{L} is interpreted as a subset of lambda-expressions that satisfies Equations 2-4 according to the correspondence stated in Equation 5. QED

For a given signature, transformation rules for applications may be derived — literally in case an internal definition actually exists and virtually in case a definition exists only in terms of some idealised Howard language.

Derivation is by *specialisation*, which is similar to mathematical projection of a function of two variables by fixing the value of one. An alternative term in programming language

research is *partial evaluation* and it carries over to logic in agreement with the Curry-Howard correspondence [3]. The advantage of this approach is that programming details of some internal definition get eliminated by specialisation. Furthermore it gives credit to the use of prototypes in system development.

It all means that every operation identifies a transformation rule for its application, and that program composition corresponds to combined use of inference rules. In this sense a program is a proof, with the only addition that recursion in a definition is an appeal to proof by induction.

DEF-schemes can be considered key axioms of a logic with proofs encoded as Howard programs. Other axioms, all briefly described in Box 1 are rather trivial.

5 Equational reasoning with presumptions

A *DEF-rule* is an instance of a given signature's DEF-scheme. Equation 1 illustrates such a DEF-rule, which is repeated here for your convenience:

$$\exists_{r\{E_D\}\{E_A\}}\theta. = \exists_{D_r}\theta. \dashv \left[\begin{array}{l} \exists_{D\{D_x\}}\theta. = \exists_{E_D}\theta. \dashv \left[\exists_x\theta. = \exists_{D_x}\theta. \right] \\ \exists_{A\{D_f\}}\theta. = \exists_{E_A}\theta. \dashv \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{D_f}\theta. \\ \dashv \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \end{array} \right] \end{array} \right] \quad (1)$$

Instantiation with $E_D=x*x$, $E_A=f\{3\}$ and $D_r=A\{D\{X*2\}\}$ we obtain:

$$\begin{array}{l} \exists_{r\{x*x\}\{f\{3\}\}}\theta. = \exists_{A\{D\{X*2\}\}}\theta. \\ \dashv \left[\begin{array}{l} \exists_{D\{D_x\}}\theta. = \exists_{x*x}\theta. \dashv \left[\exists_x\theta. = \exists_{D_x}\theta. \right] \\ \exists_{A\{D_f\}}\theta. = \exists_{f\{3\}}\theta. \dashv \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{D_f}\theta. \\ \dashv \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \end{array} \right] \end{array} \right] \end{array}$$

Presumptions are unaffected by this initial step, but now we need to instantiate the second with $D_f=D\{X*2\}$ to get a rule by which the right-hand side of the equation can be transformed:

$$\begin{array}{l} \exists_{r\{x*x\}\{f\{3\}\}}\theta. = \exists_{A\{D\{X*2\}\}}\theta. \\ \dashv \left[\begin{array}{l} \exists_{D\{D_x\}}\theta. = \exists_{x*x}\theta. \dashv \left[\exists_x\theta. = \exists_{D_x}\theta. \right] \\ \exists_{A\{D_f\}}\theta. = \exists_{f\{3\}}\theta. \dashv \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{D_f}\theta. \\ \dashv \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \end{array} \right] \\ \exists_{A\{D\{X*2\}\}}\theta. = \exists_{f\{3\}}\theta. \dashv \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{D_f}\theta. \\ \dashv \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \end{array} \right] \end{array} \right] \end{array}$$

The presumptions now consist of the two from the general rule and one instantiated from one of these. We shall for brevity memorise the former to be instantiated when needed. However, when an instantiated presumption is used, its presumptions get introduced.

$$\begin{aligned}
& \exists_{r\{x*x\}\{f\{3\}\}} \theta. \\
& = \exists_{f\{3\}} \theta. & \vdash \left[\begin{array}{c} \dots \\ \exists_{f\{E_X\}} \theta. = \exists_{D\{X*2\}} \theta. \vdash \left[\exists_X \theta. = \exists_{E_X} \theta. \right] \end{array} \right] \\
& = \exists_{D\{X*2\}} \theta. & \vdash \left[\begin{array}{c} \exists_{D\{D_X\}} \theta. = \exists_{x*x} \theta. \vdash \left[\exists_x \theta. = \exists_{D_x} \theta. \right] \\ \dots \\ \exists_X \theta. = \exists_3 \theta. \end{array} \right] \\
& = \exists_{x*x} \theta. & \vdash \left[\begin{array}{c} \dots \\ \exists_X \theta. = \exists_3 \theta. \\ \exists_x \theta. = \exists_{X*2} \theta. \end{array} \right] \\
& = \exists_{x*2} \tau_1. \exists_{x*2} \tau_2. [\theta \triangleleft (\tau_1 \cdot \tau_2)] & \vdash \left[\begin{array}{c} \dots \\ \exists_X \theta. = \exists_3 \theta. \end{array} \right]
\end{aligned}$$

Continued transformation by rules for infix operators trivially leads to the value 36 in agreement with the lambda-calculus example (page 3).

6 Language design and transformation semantics

Dijkstra has introduced *predicate transformers* as semantics of a programming language, striving to establish programming as a discipline of mathematics. Semantics had before that mostly been presented as mathematical models, with the notion of *state* as a stumble point for mathematicians. The important advantage of using transformations is elimination of state changes, rather than explaining a process in terms of a sequence of states.

Dijkstra's semantics did not include definitions. Internal definitions as introduced above depend on context and their semantics is given by DEF-schemes. So Howard languages illustrate that transformation semantics can be expressed for languages more general than Dijkstra's.

A tool exists to implement Howard languages with support for a fixed, common syntax and the following concepts:

- internal definition with signatures (of a fairly general notion of operations)
- operation applications including infix notation with operator symbols
- usual notation for sequential composition of computations
- type inference with types considered sets (in the mathematical sense)

Essentially there are so few predefined operations that the core can hardly be classified as a programming language. The tool helps introduce predefined operations from a given signature.

Support for one family of languages can be considered both negative and positive from a language designer's point of view. Language implementers are restricted syntactically but provided with strong support for type checking and modularity. The tool represents a conservative choice of syntax and allows conveniences where a strict syntactic reflection of concepts is undesirable (e.g. 'declaration' as a shorthand for an application).

6.1 Semantics of signatures

The term 'signature' is related to similar concepts in other programming languages, and some might easily and reasonably consider them essentially identical. Some consider it a simplified version of a concept not yet clarified and blame it for not covering semantics of the operation with a given signature.

Before going on, recall that internal definitions are similar to classes while an item being defined is similar to a member. Probably we agree that the semantics of classes and members should not be confused, so likewise we need to distinguish between semantics of an internal definition and the item being defined. So:

A signature encodes semantics of internal definitions+

Concretely it means that a signature can be translated into a transformation rule for possible instances of internal definitions with the given signature.

6.2 Signatures and command syntax

Signatures determine the syntax of named commands as given below in EBFN: a pair of braces contains patterns that can be iterated (or omitted), a pair of bracket contains patterns that can be omitted. Actual symbols appear in typed font between “- and ”-characters.

Signatures identified by names

Signature = Name [TypeInf] { “[” Signature “]” } [“:” Type] |
 TypeInf = “OF” [Numeral] Name { “,” [Numeral] Name }

Application syntax

Expr = Name { [Label [“:”]] Arg } | “ DEF” Signature Arg Arg
 Arg = “{” Expr { “;” Expr } “}”

The number of required arguments in an application is equal to the length of the list of signatures following the name in the operation's signature. The syntax above is simplified to emphasise the correspondence between signatures and arguments. Box 2 provides a more complete description.

Convenience rules prescribe some special notations as identical to expressions that adhere to the syntax:

Box 2 Signatures and expressions

Signature	= BasicSignature OpSignature CbVSignature
BasicSignature	= Identification { “[” Signature “]” } [“:” TypeExpression]
Identification	= Name [“OF” TypeOperator { “,” TypeOperator }]
TypeOperator	= [NaturalNumber] Name
TypeExpression	= { TypeExpression } Name
CbVSignature	= Identification ValueList { “[” Signature “]” } “:” TypeExpression
ValueList	= “(“ TypedSignature { “,” TypedSignature } “)”
TypedSignature	= Identification [ValueList] “:” TypeExpression OperatorSignature
OpSignature	= OperatorId “(“ TypeExpression “,” TypeExpression “)” “:” TypeExpression
OperatorId	= OperatorSymbol [“OF” TypeOperator { “,” TypeOperator }]
Expression	= E [Qualifier “.”] Name { Argument ArgList }
Qualifier	= LevelName
Argument	= [LevelName] “{” Expression { “;” Expression } [“;”] “}”
ArgList	= [LevelName] “(“ Expression { “,” Expression } “)”
LevelName	= Name [“:”]
E	= Application [O] [E] { [E] O E } [E] [O] “(“ E “)”
O	= OperatorSymbol

A ?-symbol is assumed, if no other operator separates two expressions.

Identity is a braced, semicolon separated lists of expressions not being an argument of an application. Identities are interpreted as arguments of a **program** operation which is a polymorphic identity.

Declaration allows an application to be written with its last argument apparently missing, but present as ‘the remaining part of a context’. This is often used to write an internal definition as in other languages, and similarly use common notation for instantiation of a class.

Syntactic coercion allows a level name to be used by itself, provided a name for a *default member* is introduced in the argument it applies to. The default member name is by convention `---`. Further, when also a name `_` is introduced, the default member name may be replaced by that when used as a call-by-value argument, or according to descriptions of operator symbols.

A **list expression** is a bracketed, comma separated lists of expressions. Such are interpreted as a `::` separated lists of the expressions terminated by `::nil`, with the operator symbol and `nil` being user defined.

Language designers who want mutable *variables* may provide an operation for their declaration with a signature like

```
var OF w, rvalue [Scope OF lvalue
    (__:lvalue)
    [_:rvalue]
    :=(lvalue,rvalue) : rvalue]:w]:w
```

Operation `var` may be applied as in `{ var x; ...; x:=x+1; ... }` which will be disambiguated as `{ var x{ ...; x.__:=x._+1; ... } }`, i.e. by taking ‘the rest of the brace’ as the missing argument in an application written as a ‘declaration’

6.3 Example: induction

An operation can be defined to match the structure of proofs by induction:

```
induction OF Problem, Result
  [Initial:Problem]
  [Break_down[__:Problem]
   [result[Sub:Problem]:Result]:Result]:Result
```

and it might be useful in exercises to teach programming with easy-to-prove iterations, e.g. to compute the product of elements in a list use

```
induction{[3,5,7]} L: { split_list{L}{1}{hd*result{tl}} }
```

7 Specialisation

A well-known example, `twice`, can be internally defined and used as follows

```
{ DEF twice [F{x:int}:int] [Return[f{X:int}:int] :int]:int
  { Return{F{F{X}}}} };
  twice{x*x}; # i.e. F is x -> x*x with implicit x
  f{2}       # so f(2) is (2*2)*(2*2)
}
```

So, although only values can be returned, it is possible to provide access to a function as a member, say ‘`f`’, of a class-like operation. The example uses the syntactic convenience rule for ‘declarations’ that makes it equivalent to

```
{ DEF twice [F{x:int}:int] [Return[f{X:int}:int] :int]:int
  { Return{F{F{X}}}} }
  { twice{x*x}{f{2}} }
}
```

i.e. the apparently missing argument is taken as ‘the rest of the brace’. In this form there is no need for the outermost braces.

It is bad to restrict the result of the entire computation to `int`. but the type systems allows the following which calls for type inference to bind the type name `W` appropriately.

```
{ DEF twice OF W [F[x:int]:int] [Return[f[X:int]:int] :W]:W
  { Return{F{F{X}}}} };
  twice{x*x}; # i.e. F is x -> x*x with implicit x
  {f{2}}      # so f{2} is 2-> {2*2}*{2*2}
}
```

in this case with the same result, but now `f{2}` can be replaced by something like `stdout << f{2} << nl` of type `Output`.

`INTERPRET` is the name of a special operation that interprets an expression in context, as for instance in a nested read-eval-print-loop

```
DEF twice OF W [F[x:int]:int] [Return[f[X:int]:int] :W]:W
  { Return{F{F{X}}}} }
  { twice{x*x}          # i.e. F is x -> x*x with implicit x
    {loop{INTERPRET}} # expresses a read-eval-print-loop
  }
```

so that an input of `f{2}` will return 16.

An operation, `defrule`, defined internal to the interpreter can be used to generate \LaTeX -text for inclusion in documentations, and in this case `defrule{"twice"}` has been used to for the manuscript to include:

$$\begin{array}{l}
\text{twice OF W} \\
\quad [F[x:\text{int}]:\text{int}] \\
\quad [Return[f[X:\text{int}]:\text{int}]] \\
\exists_{\text{twice}\{E_F\}\{E_{Return}\}}\theta. = \exists_{D_{\text{twice}}}\theta. \\
\quad \neg \left[\begin{array}{l} \exists_{F\{D_x\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_{D_x}\theta. \right] \\ \exists_{Return\{D_f\}}\theta. = \exists_{E_{Return}}\theta. \neg \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{D_f}\theta. \\ \neg \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \end{array} \right] \end{array} \right]
\end{array}$$

7.1 Details: Specialising operation twice

The computational definition of `twice` is known and can be used to derive a rule for application of `twice`. The idea is similar to the projection of a function of pairs of values by keeping one fixed.

Substitution of `Return{F{F{X}}}` for D_{twice} and the consequential `F{F{X}}` for D_f gives

$$\begin{array}{l}
\exists_{\text{twice}\{E_F\}\{E_{Return}\}}\theta. = \exists_{Return\{F\{F\{X\}\}}}\theta. \\
\quad \neg \left[\begin{array}{l} \dots \\ \exists_{Return\{F\{F\{X\}\}}}\theta. = \exists_{E_{Return}}\theta. \neg \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{F\{F\{X\}\}}\theta. \\ \neg \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \end{array} \right] \end{array} \right]
\end{array}$$

Recall that elipsis denote memorised presumptions. We can instantiate the first presumption twice, once with $F\{X\}$ and once with X substituted for D_X :

$$\begin{aligned} \exists_{\text{twice}\{E_F\}\{E_{\text{Return}}\}}\theta. &= \exists_{\text{Return}\{F\{F\{X\}\}\}}\theta. \\ \neg \left[\begin{array}{c} \dots \\ \exists_{F\{F\{X\}\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_{F\{X\}}\theta. \right] \\ \exists_{F\{X\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_X\theta. \right] \end{array} \right] \end{aligned}$$

The two rules can now be combined to one as in

$$\begin{aligned} \exists_{\text{twice}\{E_F\}\{E_{\text{Return}}\}}\theta. &= \exists_{\text{Return}\{F\{F\{X\}\}\}}\theta. \\ \neg \left[\begin{array}{c} \dots \\ \exists_{F\{F\{X\}\}}\theta. = \exists_{E_F}\theta. \neg \left[\begin{array}{c} \exists_x\theta. = \exists_{F\{X\}}\theta. \\ \exists_{F\{X\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_X\theta. \right] \end{array} \right] \end{array} \right] \end{aligned}$$

This rule can be simplified by use of the third presumption to

$$\begin{aligned} \exists_{\text{twice}\{E_F\}\{E_{\text{Return}}\}}\theta. &= \exists_{E_{\text{Return}}}\theta. \\ \neg \left[\begin{array}{c} \dots \\ \exists_{F\{F\{X\}\}}\theta. = \exists_{E_F}\theta. \neg \left[\begin{array}{c} \exists_x\theta. = \exists_{F\{X\}}\theta. \\ \exists_{F\{X\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_X\theta. \right] \end{array} \right] \end{array} \right] \end{aligned}$$

The bottom two presumptions can be combined to eliminate $\exists_{F\{F\{X\}\}}\theta.$, and at the same time drop the two presumptions at the top:

$$\begin{aligned} \exists_{\text{twice}\{E_F\}\{E_{\text{Return}}\}}\theta. &= \exists_{E_{\text{Return}}}\theta. \\ \neg \left[\begin{array}{c} \exists_{f\{E_X\}}\theta. = \exists_{E_F}\theta. \neg \left[\begin{array}{c} \exists_X\theta. = \exists_{E_X}\theta. \\ \exists_x\theta. = \exists_{F\{X\}}\theta. \\ \exists_{F\{X\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_X\theta. \right] \end{array} \right] \end{array} \right] \end{aligned}$$

Finally the presumptions of the presumption can be combined so that every explicit reference to the defining context gets eliminated:

$$\begin{aligned} \exists_{\text{twice}\{E_F\}\{E_{\text{Return}}\}}\theta. &= \exists_{E_{\text{Return}}}\theta. \\ \neg \left[\begin{array}{c} \exists_{f\{E_X\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \right] \end{array} \right] \end{aligned}$$

Applications of operation f in E_{Return} is the given by the rule

$$\exists_{f\{E_X\}}\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_{E_F}\theta. \neg \left[\exists_x\theta. = \exists_{E_X}\theta. \right] \right]$$

So for any `int`-constant c we have

$$\begin{aligned}
& \exists_{\mathbf{f}\{c\}} \theta. \\
&= \exists_{\mathbf{x}*\mathbf{x}} \theta. \dashv \left[\exists_{\mathbf{x}} \theta. = \exists_{\mathbf{x}*\mathbf{x}} \theta. \dashv \left[\exists_{\mathbf{x}} \theta. = \exists_c \theta. \right] \right] \\
&= \exists_{\mathbf{x}} \tau_1. \exists_{\mathbf{x}} \tau_2. [\theta \triangleleft (\tau_1 \times \tau_2)] \dashv \left[\exists_{\mathbf{x}} \theta. = \exists_{\mathbf{x}*\mathbf{x}} \theta. \dashv \left[\exists_{\mathbf{x}} \theta. = \exists_c \theta. \right] \right] \\
&= \exists_{\mathbf{x}*\mathbf{x}} \tau_1. \exists_{\mathbf{x}*\mathbf{x}} \tau_2. [\theta \triangleleft (\tau_1 \times \tau_2)] \dashv \left[\exists_{\mathbf{x}} \theta. = \exists_c \theta. \right] \\
&= \exists_{\mathbf{x}} \mu_1. \exists_{\mathbf{x}} \mu_2. [\tau_1 \triangleleft (\mu_1 \times \mu_2)] \exists_{\mathbf{x}} \mu_5. \exists_{\mathbf{x}} \mu_6. [\tau_2 \triangleleft (\mu_3 \times \mu_4)] [\theta \triangleleft (\tau_1 \times \tau_2)] \\
&\quad \dashv \left[\exists_{\mathbf{x}} \theta. = \exists_c \theta. \right] \\
&= \exists_c \mu_1. \exists_c \mu_2. [\tau_1 \triangleleft (\mu_1 \times \mu_2)] \exists_c \mu_5. \exists_c \mu_6. [\tau_2 \triangleleft (\mu_3 \times \mu_4)] [\theta \triangleleft (\tau_1 \times \tau_2)] \\
&= [\theta \triangleleft (c \times c) \times (c \times c)]
\end{aligned}$$

8 Sideeffects

While a program is being developed, programmers sometimes need to add code to reflect progress (read: problematic behaviours). Mostly it amounts to what is known as ‘side-effects’ because it really illustrates that programs differ from mathematical expressions in kind.

An illustration from an application of operation `twice` with a read-eval-print-loop is

```
.demo 1> {var t; t:=0; f{stdout << ~"Step " << t:=t+1 << nl; 2} };
Step 1
Step 2
Step 3
Step 4
16
```

which reveals that the argument of operation `f` is ‘computed’ four times to arrive at the result 16, as postulated earlier.

This is actually reflected in the rewriting to evaluate $\mathbf{f}\{c\}$ in the previous section: the step prior to the last contains $\exists_c \mu_k.$ for $k=1-4$. Obviously this is a point where efficiency can become important, and hence makes a need for means to avoid such repetitions. In other words: a need for call-by-value arguments.

Arguments for which no name is introduced implicitly are the points where the issue of such side-effects may arise. The term *call-by-name* is used when such an argument is computed whenever the interpreter refers to it. A companion term is *call-by-value* which implies that a single reference is used to obtain the argument’s value, which is then saved for later use.

The complete syntax of signatures allows call-by-value to be requested, but notice that it is a programmer’s responsibility to meet such requests. The actual syntax is simply to allow parentheses instead of brackets around a typed signature. In the example all brackets could be replaced by corresponding parentheses. Operation `with` evaluates its first argument as a value and passes it to its second argument.

The following definition and its use illustrates how call-by-value can be achieved

```

2> DEF twice OF W [F(x:int):int] [Return[f(X:int):int] :W] : W
3>   { Return{with(X) u; F{F{u}}}} }
4>   { twice{x*x}          # i.e. F is x -> x*x with implicit x
5>     {loop{INTERPRET}} # similar to loop-eval-print
6>   };
. 1> {var t; t:=0; f{stdout << ~"Step " << t:=t+1 << nl; 2} };
Step 1
16

```

More serious problems are that requests for call-by-value should be reflected in the semantics of an internal definition, and that means to express it should be found. The problems have been solved, as illustrated in the following subsection.

8.1 Call-by-value vs. call-by-name

Operation `defrule("twice")` generates the manuscript for the following display after the signature has been changed:

$$\begin{array}{l}
\text{twice OF W} \\
[F(x:\text{int}):\text{int}] \\
[\text{Return}[f(X:\text{int}):\text{int}]:W]:W \\
\exists_{\text{twice}\{E_F\}\{E_{\text{Return}}\}}\theta. = \exists_{D_{\text{twice}}}\theta. \\
\neg \left[\begin{array}{l} \exists_{F\{D_x\}}\theta. = \exists_{D_x}\eta_1. [x \triangleleft \eta_1] \exists_{E_F}\theta. \\ \exists_{\text{Return}\{D_f\}}\theta. = \exists_{E_{\text{Return}}}\theta. \neg \left[\exists_{f\{E_X\}}\theta. = \exists_{E_X}\eta_1. [X \triangleleft \eta_1] \exists_{D_f}\theta. \right] \end{array} \right]
\end{array}$$

First note the use of parentheses instead of brackets in the signature. Second note that D_x on the right-hand side in the conclusion of the first presumption tells that the mathematical expression bound to the η_1 is substituted for x once and that such an expression cannot contain side-effects. Of course this implies that the previous description of the $\text{---} \stackrel{\text{def}}{=} \{\text{---}\}$ is incomplete and the complete version has been implemented in the tool.

It makes a nice exercise for readers to specialise the above rule for definition of `twice` to a rule for applications of `twice` until the following point is reached:

$$\begin{array}{l}
\exists_{\text{twice}\{x*x\}\{f\{c\}\}}\theta. = \exists_{f\{c\}}\theta. \\
\neg \left[\begin{array}{l} \exists_{f\{E_X\}}\theta. = \exists_{E_X}\eta_1. [X \triangleleft \eta_1] \exists_{F\{F\{X\}\}}\theta. \\ \exists_{F\{F\{X\}\}}\theta. = \exists_{F\{X\}}\eta_1. [x \triangleleft \eta_1] \exists_{x*x}\theta. \\ \exists_{F\{X\}}\theta. = \exists_X\eta_1. [x \triangleleft \eta_1] \exists_{x*x}\theta. \end{array} \right]
\end{array}$$

The next step is to use the presumptions by equational reasoning:

$$\exists_{f\{c\}}\theta. = \exists_c\eta_1. [X \triangleleft \eta_1] \exists_{F\{F\{X\}\}}\theta. = \exists_c\eta_1. [X \triangleleft \eta_1] \exists_{F\{X\}}\eta_1. [x \triangleleft \eta_1] \exists_{x*x}\theta.$$

Now we face a situation where substitution is tricky. The η_1 in $[X \triangleleft \eta_1]$ is a free occurrence that will become bound if the substitution is done thoughtlessly, since every occurrence

of η_1 is bound by $\exists_{F\{x\}}\eta_1..$ The problem is overcome by renaming (i.e. the rule called α -conversion in the lambda-calculus). Thus we may proceed

$$\begin{aligned}
\cdots &= \exists_c \eta_2. [X \triangleleft \eta_2] \exists_{F\{x\}} \eta_1. [x \triangleleft \eta_1] \exists_{x*x} \theta. \\
&= \exists_c \eta_2. [X \triangleleft \eta_2] \exists_x \eta_3. [x \triangleleft \eta_3] \exists_{x*x} \eta_1. [x \triangleleft \eta_1] \exists_{x*x} \theta. \\
&= \exists_c \eta_2. [X \triangleleft \eta_2] \exists_x \eta_3. [x \triangleleft \eta_3] [\eta_1 \triangleleft (x \times x)] [x \triangleleft \eta_1] [\theta \triangleleft (x \times x)] \\
&= \exists_c \eta_2. [X \triangleleft \eta_2] \exists_x \eta_3. [x \triangleleft \eta_3] [\theta \triangleleft ((x \times x) \times (x \times x))] \\
&= \exists_c \eta_2. [X \triangleleft \eta_2] [\theta \triangleleft ((X \times X) \times (X \times X))] \\
&= [\theta \triangleleft ((c \times c) \times (c \times c))]
\end{aligned}$$

Enough details have been presented so that a careful reading will show where renaming is required to avoid unintended name bindings.

9 Verification

Program verification implies application of the transformation rules along with some expression that may be a predicate. Rules can be identical to familiar rules introduced by others, e.g. Hoare's rule for assignments. A rule for use of operation **induction**, cf. page 12 matches proof by mathematical induction. Side-effects are essentially described like assignments to hidden variables.

Two examples follow to illustrate (a) elimination of program variables and (b) informal verification based on equivalence of program fragments.

Elimination

Hoare's rule for assignments is: $\exists_{x:=E} \theta. = \exists_E \theta. [x \triangleleft \theta]$. Right-to-left progress implies that an expression using variable x in the state just after its initialisation can be quite complex but have the form \mathcal{E}_x . Initialisation implies that x gets substituted by some expression, often a constant, so that x gets eliminated from the resulting mathematical expression: $[x \triangleleft e] \mathcal{E}_x$.

Induction

Mathematical induction proves

$$(\exists_{\text{induction}(n) \ i: \{\text{if}(i=0, 0(2*i-1)+\text{result}(i-1))\}} \theta. \theta) = \sum_{i=0}^n (2i - 1)$$

Equivalence

A slight variation of the previous example is

$$\exists_{\text{var } x} \{x:=0; \text{induction}(n) \ i: \{\text{if}(i=0, x) \{\text{result}(i-1); x:=2*i-1\}\}; y:=x\} \theta. \mathcal{P}$$

which transforms to

$$[x < 0] \exists_{\text{induction}(n) \ i: \{\text{if}(i=0, x) \{\text{result}(i-1); x:=x+2*i-1\}\}} \theta. \exists_x \theta. [y < \theta] \mathcal{P} \quad (6)$$

With the presumption as induction hypothesis we can prove

$$\begin{aligned} \exists_{\text{result}(k)} \theta. &= \exists_{\text{induction}(k) \ i: \{\text{if}(i=0, x) \{\text{result}(i-1); x:=x+2*i-1\}\}} \theta. \\ \dashv \left[\exists_{\text{result}(i)} \theta. &= \left[\theta < (x + \sum_{j=0}^k (2j-1)) \right] \dashv \left[i < k \right] \right] \end{aligned}$$

and conclude:

$$\forall n. \exists_{\text{induction}(n) \ i: \{\text{if}(i=0, x) \{\text{result}(i-1); x:=x+2*i-1\}\}} \theta. \theta = x + \sum_{i=0}^n (2i-1)$$

The entire expression 6 then becomes

$$[x < 0] [\theta < (x + \sum_{i=0}^n (2i-1))] [x < \theta] [y < \theta] \mathcal{P}$$

which finally can be reduced to

$$[\theta < \sum_{i=0}^n (2i-1)] [x < \theta] [y < \theta] \mathcal{P}$$

so x is eliminated but its final value may survive elsewhere in the context.

A more efficient computation results, if application of **induction** is replaced by an equivalent application of **while**. The rule for the former is the familiar and obvious, and the path followed here might be useful in other situations.

Detailed inference rules have not been presented, for brevity and because this should not be considered an advocacy for specific operations. But hopefully the examples are illustrative anyway.

9.1 Partial and total correctness

With rules as presented, programs as proofs can justify partial correctness only, i.e. proofs presume program termination. So a separate proof of termination is required.

With generated transformation rules, termination is ensured only if every program step terminates, so one may have to add manually presumptions to ensure termination. Sometimes that requires knowledge about a specific application domain.

Signatures may exist for operations that belong to a specific field of applications. An example is presented later and concerns unix-style process control (see page 20). Termination properties with the chosen ‘members’ are complex and it may be impossible to reduce them to the usually known techniques for proofs of termination. A better design may, perhaps, encapsulate the problem.

Howard languages are adequate as domain specific languages, and the issue of termination has to be delegated to specialist from the application field. How to help a specialist design operations with appropriate termination properties is an open question.

10 Components and their composition

The notion of a component in this context is delightfully easy and general:

A component is a named operation with a signature

It means that the notion of a named operation is very broad, covering simple functions, structured statements, class-like entities, members that are operations, and generalised classes with several arguments each with own members.

Composition is the same for all kinds of operations and corresponds to function composition in the simplest case and statement composition otherwise. *Implicit name introduction* and the simple notion of types as sets have been essential for this simplification. Sets of functions are thus not admitted.

Transformation rules for internal definitions are expressed in terms of unknown program fragments. Each application implies a binding of unknowns to actual fragments and thus makes specialisation possible. Transformation and evaluations work in opposite directions: the former right-to-left the latter left-to-right.

Design of components can be a complex task. But a strict regime as represented by the requirement of a signature (as presented) for every component tends to help in design.

One thing is components of a single program, another components of collections of programs. Experiences tells that a collections of programs may be connected by channels, e.g. POSIX pipes. Texts sent over a channel can be interpreted as program expressions in a context at the receiving side. One important predefined operation supports this behaviour in an actual interpreter.

10.1 UNIX-based control operations

An operation `unix` has been implemented with a signature shown in Box 3. The names `PIPE` and `PID` are member names of types, i.e. fresh types that should not be confused with others, not even the same in another `unix`-application.

Member `run` is an example of a complex member that can be used class-like. It is used to invoke programs in separate processes and connect them in various ways, e.g. with pipes or to files. In terms of low-level primitives, the second argument of `run` has to describe how file descriptors are used as ports for interconnections.

A typical programming error with UNIX which prevents termination of a process control program, is failure to close a pipe for which a controlled process expects an end-of-file condition to arise. A designer may strive to define a complex operation so that disciplines to guard against such errors are enforced.

10.2 Sorting

It has been claimed that operations may be class like, and that the notion of classes may be generalised and have more than a single argument where members are introduced. It seems not to be widely needed but please note:

Box 3 Signature of operation `unix`

```
unix
  [Members OF PIPE, PID
    [newpipe:PIPE]
    [child[Init[exec(arg:string Array)]]:PID]
    [mk_stdin(x:PIPE)]
    [mk_n(x:PIPE)]
    [mk_stderr(x:PIPE)]
    [source OF W
      (x:PIPE)
      [Access(in:Input, __:Input):W]:W]
    [dest OF W
      (x:PIPE)
      [Access(out:Output, __:Output):W]:W]
    [close(x:PIPE)]
    [kill(p:PID)]
    [await(id:PID):int]
    [await_all]
    [run(Cmd:string Array)
      [Con OF IO
        [<(NONE,string) : IO]
        [>(NONE,string) : IO]
        [>>(NONE,string) : IO]
        [<(NONE,PIPE) : IO]
        [>(NONE,PIPE) : IO]
        [<(NONE,Input) : IO]
        [>(NONE,Output) : IO]:IO List]]]
```

- Some languages have a predefined operation `sort` that distinguish an input and an output phase. An operation can be defined with an argument for each phase, one with an `put`-operation, another with an operation to receive elements after ordering:

```
sort OF T,W
  [ InOrder [x:T] [y:T]:int ]
  [ Send [put[X:T]] ]
  [ Receive [all[Body[x:T]]]:W ] :W
```

The first argument allows users to specify an ordering, e.g. as `x<y`.

- There is a nice symmetry between the definition and application parts of an internal definition. So an internal definition can itself be considered an operation with two arguments with different ‘member names’.

- Class-like operations are defined in terms of an argument name that defines meanings of members by an application of a name to many arguments. An example is the **Scope**-argument of operation **var**. A virtual internal definition would use the pattern (with labels as reminders before each argument):

Scope __ { ... } _ { ... } asg { ... }

11 Conclusions

The verb ‘to prove’ is transitive, but it is possible to conceive the noun ‘proof’ independently of an object as required by the verb. A proof can be understood as a pattern of references to individual rules of a formal logic. In that sense a program can be seen as a proof provided each construct used in the composition of a program has a unique proof-rule associated with it. A foundation to build a programming language with just such operations has been presented.

A benefit of programs as proofs is that not all programmers need to master the logic aspects of programs completely. One may still acknowledge programmers work as proofs in a substantial sense. Designers should, of course, be more concerned with the rules for application of their operations, i.e. their adequateness as axioms of a logic.

Another benefit might be the implicit enforcement of a programming discipline that is indirectly influenced by formal logic, much the same as discussions in daily life may benefit from rules justified by formal logic.

Howard languages have operations that identify axioms, even if not explicitly expressed in a description. Hence one might extend the Curry-Howard correspondence by considering a specific Howard language as a theory.

A traditional proof of a program is merely a proof check that the given proof is effective in a given context. Failure might stem from an incomplete case analysis, and that might lead to useful feed-back to programmers.

A programming language with constructs that all have signatures and hence possibly an internal definition rightly deserves to be called modular. New constructs can be implemented independently and added to extend existing languages. So languages and their semantics can be built incrementally.

With the very broad notion of operations, it is easy to develop components by separate teams, if only they agree on component signatures. The intended semantics can and should be negotiated between teams from the outset and revised during development with cost estimates of changes.

Concepts about programming languages, semantics, and logic in the sense of transformation rules stem from previous work on a statement-oriented approach to data abstraction [6]. For programming practice it seems to be a realistic realisation of programs as proofs.

Experimental tools have been developed to support programming language design, interpreter construction, and formalised documentation of semantics of operations. The tools also help to incrementally implement and install new operations with given signatures as

predefined entities.

The programs-as-proofs aspect of the Curry-Howard correspondence is corroborated by this work. The propositions-as-types aspect only to a lesser degree. One can illustrate this with an expression outline

$$\exists_{\mathbf{p}} \theta : T.R_{\theta}$$

making it clear that T and R_{θ} have different rôles. Although it might be tempting to have T express the set of primes (or even subsets of primes) it seems simpler to keep the notions apart. Not the least with regard to acceptance by programmers.

Mathematicians rarely rely on fully formal proofs, but formally state their goals. So perhaps one might say: programmers rarely rely on formally stated goals, but formally state their proofs as programs.

References

- [1] Graham M. Birtwistle, Ole-John Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur, 1980.
- [2] E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [3] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. P-H, 1993.
- [4] P. Martin-Löf. Constructive mathematics and computer programming. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, International Series in Computer Science, pages 167–184. Prentice-Hall, 1985.
- [5] Peter Naur. Revised report on the algorithmic language Algol 60. . *Communications of the ACM*, 6(1):1–17, 1963.
- [6] J. Steensgaard-Madsen. A statement-oriented approach to data abstraction. *ACM Transactions on Programming Languages and Systems*, 3(1):1–10, January 1981.
- [7] Jørgen Steensgaard-Madsen. Pascal — clarifications and recommended extensions. *Acta Informatica*, 12(1):73–94, 1979.